Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Simon Gerhold

Razvoj interaktivne Rubikove kocke

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Rok Rupnik

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00562 / 2013
Datum: 5.11.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **SIMON GERHOLD**

Naslov: **RAZVOJ INTERAKTIVNE RUBIKOVE KOCKE**
THE DEVELOPMENT OF INTERACTIVE RUBIK'S CUBE

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Zasnujte optimalen način predstavitve Rubikove kocke v objektno usmerjenem jeziku s pomočjo razredov in povezav med njimi ter prikaz kocke v trodimenzionalnem prostoru. Za grafični vmesnik uporabite tehnologijo WPF (Windows Presentation Foundation). Razvijte namizno aplikacijo, kjer lahko uporabnik s pomočjo miške izvaja vse možne rotacije nad kocko. Aplikacija naj omogoča tudi animacijo prikaza možnosti rešitve Rubikove kocke po metodi Fridrich. Rešitev naj bo razdeljena na več korakov, pri čemer naj bo vsak posamezen korak predstavljen opisno in v obliki animacije.

Mentor:

doc. dr. Rok Rupnik

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Simon Gerhold, z vpisno številko **63010028**, sem avtor diplomskega dela z naslovom:

Razvoj interaktivne Rubikove kocke

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Roka Rupnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 4. marca 2014 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Rok Rupniku za nasvete in pomoč pri izdelavi diplomske naloge.

Posebna zahvala pa gre punci Jasmini za potrpežljivost in podporo pri zaključku moje študijske poti.

Kazalo

Slike

Povzetek

Rubikova kocka je ena najbolj znanih igrač na svetu, saj igralcu kljub enostavnim pravilom predstavlja zapleten problem. Ker je vseh možnih permutacij nekaj več kot 43 trilijonov, je rešitev vse prej kot trivialna zadeva. Prav zaradi te kompleksnosti je Rubikova kocka predmet številnih raziskav v računalništvu, še posebej v zadnjih letih, ko se zdi, da smo s pomočjo super zmogljivih računalnikov problem končno obvladali.

V diplomskem delu je bil poglavitni cilj predstavitev Rubikove kocke v objektno usmerjenem jeziku s pomočjo razredov in povezav med njimi ter prikaz kocke v trodimenzionalnem prostoru. Za grafični vmesnik se je uporabila tehnologija WPF (Windows Presentation Foundation), ki že privzeto nudi velik nabor gradnikov za delo z interaktivno grafiko. Končni rezultat je namizna aplikacija, kjer lahko uporabnik s pomočjo miške rotira celo kocko ali plast v vseh možnih smereh. Kot dodatna funkcionalnost se je dodala možnost vnosa poljubne kocke, kontrola vhodnih podatkov in prikaz rešitve s pomočjo ene najbolj znanih metod za reševanje Rubikove kocke, metode Fridrich. Rešitev je razdeljena na več korakov, pri čemer je vsak posamezen korak predstavljen opisno in v obliki animacije.

Ključne besede: Rubikova kocka, metoda Fridrich, objektno usmerjeno programiranje, 3D grafika, funkcionalni testi

Abstract

Rubik's Cube is one of the most known toys in the world, mostly because of its simple rules, yet it represents quite a complex task to solve it. It has more than 43 quintillion possible permutations, so solving the cube is not a trivial matter. Because of its complexity, the Rubik's Cube is a subject of numerous researches in the field of computer science, especially in last couple of years, when it seems that the problem has finally been manageable.

The main goal in the thesis was to represent the Rubik's Cube in the object oriented programming paradigm with the help of classes and relationships between them and also to visually represent the cube in the 3D environment. For the graphics interface we used WPF technology (Windows Presentation Foundation), which offers a vast array of classes for interactive graphics. The final result is a desktop application, which allows the user to rotate the whole cube or just a layer in all possible directions with the help of a computer mouse. We also added the functionality of user defined input of any cube, validation of input data and solving it with the help of one of the most known methods for solving the Rubik's cube, Fridrich method. The solution is composed of multiple steps, where each step in the solution is represented descriptively and as an animation on the screen.

Key words: Rubik's cube, Fridrich method, object oriented programming, 3D graphics, unit testing

Poglavje 1

Uvod

Rubikova kocka je ena izmed najbolj znanih igrač na svetu, saj je poceni, lahko dostopna in igralcu kljub enostavnim pravilom predstavlja zapleten izziv. Rešitev kocke namreč ni trivialna, še posebej pri zadnjih korakih, kjer je potrebno v pravilno pozicijo postaviti zadnje kockice, ne da bi uničili prej pravilno rešeno strukturo.

Cilj diplomske naloge je bil pripraviti ustrezne strukture za predstavitev kocke v objektno usmerjenem programiranju, pripraviti programski vmesnik, preko katerega lahko kocko manipuliramo in pa vizualna predstavitev kocke v 3D prostoru. Ena od zahtev je bila tudi funkcionalnost, da nam aplikacija vsako kocko reši in rešitev opiše v korakih, med katerimi se lahko poljubno sprehajamo in jih opazujemo v obliki animacij. Kot rezultat diplomskega dela je nastala interaktivna aplikacija, ki uporabniku nudi igranje s kocko, vnos poljubne kocke in prikaz rešitve.

Struktura diplomskega dela:

- Drugo poglavje opisuje domeno izbranega problema - mehansko zgradbo Rubikove kocke, standardno izrazoslovje, osnovne matematične lastnosti in načine za reševanje, ki se uporabljajo tako pri začetnikih, kot tudi pri profesionalnih igralcih.

- V tretjem poglavju so opisana orodja, ki smo jih pri delu uporabili.
- Četrto poglavje je glavni del diplomske naloge, kjer je natančno opisana struktura kocke, predstavljena v programskem jeziku C#, opis izdelane aplikacije in postopek, preko katerega smo si pripravili vse potrebne algoritme za rešitev kocke.
- V petem poglavju smo opravili pregled nad rezultati dela, opisali pomanjkljivosti in ponudili morebitne dodelave aplikacije.

Poglavje 2

Rubikova kocka

2.1 O Rubikovi kocki

Rubikova kocka je mehanska igrača, sestavljena iz manjših kockic različnih barv, v najbolj pogosti izvedbi so to bela, modra, oranžna, zelena, rdeča in rumena. Njena zasnova omogoča poljubno vrtenje posameznih plasti v različnih smereh, s čimer se začetna postavitev in orientacija kockic med seboj premeša. Cilj igralca je, da s pravilno kombinacijo rotacij kocko spravi v stanje, v katerem so vse ploskve na kocki enako obarvane. V diplomskem delu se bomo omejili na standardno kocko dimenzije 3x3x3 (tri plasti manjših kockic, vsaka plast ima devet kockic).

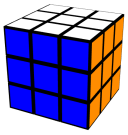
Rubikovo kocko, v obliki kot jo poznamo danes, je leta 1974 zasnoval Ernő Rubik, madžarski izumitelj in profesor arhitekture. Ideja za kocko, ki bi omogočala vrtenje svojih elementov v poljubnih smereh, je sicer obstajala že prej, vendar so bili vsi poizkusi za izdelavo uporabne verzije bolj jalovi. Ernö Rubik pa je uspel zasnovati kocko, ki je enostavna, robustna in poceni za izdelavo.

Nova igrača, ki je kljub svojim enostavnim pravilom igralcu postavila velik izziv, je hitro postala prodajna uspešnica, najprej na Madžarskem, nato še po

celem svetu. Že leta 1982 je bilo prvo mednarodno tekmovanje v hitrostnem reševanju kocke, kjer se pomerijo najboljši igralci iz celega sveta. Najboljši čas takrat je bil 22,95 sekund, lestvica pa se s časom pomika vedno višje. Trenutni uradni svetovni rekord za standardno kocko 3x3x3 je 5,55 sekunde, postavil pa ga je nizozemec Mats Valk. Poleg običajnega tekmovanja, kjer je cilj najkrajši čas rešitve kocke, pa obstajajo tudi druge kategorije, kot na primer 'slepo reševanje', kjer ima igralec zavezane oči, reševanje samo z eno roko, tekmovanje robotov itd.

Izrazoslovje, ki ga bomo uporabljali v diplomskem delu:

- Kocka: standardna kocka je dimenzije 3x3x3, sestavljena je iz 26 manjših kockic, pri čemer je namesto notranje kockice mehanizem, ki omogoča vrtenje plasti.

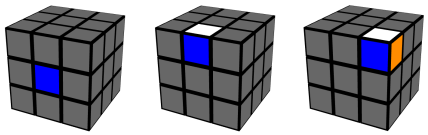


Slika 2.1: Kocka

- Barva: v standardni izvedbi ena izmed šestih možnih barv iz nabora bela, modra, oranžna, zelena, rdeča, rumena.

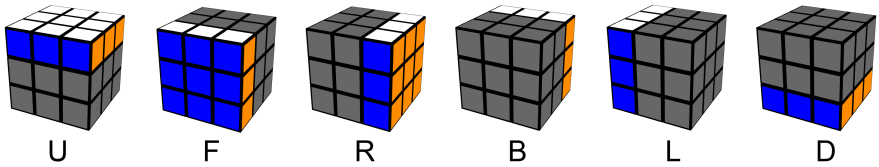
- Kockica: najmanjši sestavni cel kocke, poznamo 3 vrste:

  - sredinska kockica: kockica na sredini vsake plasti, vidna je le ena plast, pri čemer ima vsaka kockica drugačno barvo,
  - kotna kockica: kockica na stičišču dveh plasti, ki ima vidni dve plasti,
  - robna kockica: kockica na stičišču treh plasti, ki ima vidne tri plasti.



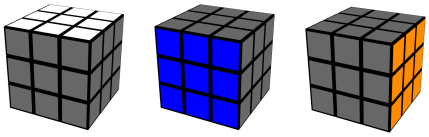
Slika 2.2: Kockice (sredinska, kotna, robna)

- Polje: plast kockice s točno določeno barvo.
- Plast: tretjina kocke, sestavljena iz devetih kockic. Standardna rubikova kocka 3x3x3 ima 3 plasti.



Slika 2.3: Notacija plasti

- Ploskev: celotna stran kocke, sestavljena iz devetih polj.



Slika 2.4: Vidne ploskve Rubikove kocke (U, F, R)

- Rotacija: predstavlja obrat celotne kocke, dveh plasti ali le ene plasti. Notacija rotacije v smeri urinega kazalca predstavlja črka plasti ali kocke, ki jo vrtimo. Pri rotaciji v nasprotni smeri urinega kazalca se črki doda pripona ', pri rotaciji za 180 stopinj pa 2.
- Algoritem: kombinacija ene ali več rotacij.
- Prstan: predstavlja 12 polj, ki obdajajo eno ploskev.

Slika 2.5: Prstan

- Križ: 5 poravnanih kockic na eni ploskvi v obliki križa.

Slika 2.6: Križ

Standardna Rubikova kocka dimenzije 3x3x3 ima 8 kotnih, 12 robnih in 6 sredinskih kockic.

Za izračun vseh možnih kombinacij kotnih kockic si prestavljajmo, da je kocka brez kotnih kockic. V prvi kot lahko damo katerokoli od osmih kockic, v drugi rob eno od preostalih sedem, v tretjega eno od ostalih šest,... Število možnih postavitev kockic je torej 8!. Vsaka od kotnih kockic pa je lahko orientirana v eni izmed treh smeri, zato se število vseh možnih kombinacij poveča za faktor 3^8. Na podoben način lahko izračunamo število možnih kombinacij kotnih kockic. Vseh možnih postavitev je 12!, vsaka pa je lahko orientirana v eni izmed dveh smereh, torej 2^{12}. To teoretično število pa je v praksi manjše - z vsemi možnimi rotacijami je možno sestaviti tretjino vseh možnih kombinacij kotnih kockic in četrtino robnih. Enačba za skupno število vseh možnih kombinacij kocke je tako

$$\frac{(8! * 3^8 * 12! * 2^{12})}{(3 * 2 * 2)} = 43.252.003.274.489.856.000.$$

2.2 Metode za reševanje

Za reševanje Rubikove kocke obstaja več metod, ki se med seboj razlikujejo predvsem po zahtevnosti. V nadaljevanju si bomo pogledali tri najpogostejše.

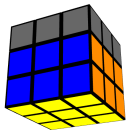
2.2.1 Metoda Fridrich

Fridrich velja za najstarejšo in najbolj znano metodo reševanja Rubikove kocke, saj se je začela razvijati že leta 1980 in je plod akumuliranega znanja večih neodvisnih igralcev. Ime je dobila po čehinji Jessici Fridrich, ki je izpopolnila algoritme in jo dodelala v danes poznano obliko. Primerna je tako za začetnike, kot tudi za profesionalne igralce na tekmovanjih. V povprečju za rešitev kocke zadostuje 55 rotacij, poznati pa je potrebno okrog 120 algoritmov. V diplomskem delu bomo predstavili najpogosteje uporabljene korake in algoritme, obstaja pa jih seveda precej več. Postopek reševanja je razdeljen v več korakov, ki se morajo izvesti pravilnem zaporedju:

1. Križ: prvi korak je pravilna postavitev in orientacija robnih kockic na eni ploskvi. Vsaka robna kockica se mora ujemati z obema sredinskima kockicama na povezanih ploskvah. V večini primerov se križ rešuje na spodnji ploskvi, da ni potrebno izgubljati časa s kasnejšim obračanjem cele kocke. Za križ potrebujemo štiri kockice, vsaka je lahko na enem od dvanajstih mest in je lahko v eni izmed dveh možnih orientacij. Število vseh možnih kombinacij je tako $12 * 11 * 10 * 9 * 2^4 = 190.080$. Križ se večinoma rešuje intuitivno, saj je tako lažje in hitrejše kot pa z učenjem algoritmov na pamet. Dokazano je, da je križ mogoče sestaviti iz poljubne začetne kocke z največ osmimi rotacijami.

Slika 2.7: Križ

2. Prvi dve plasti (F2L - First Two Layers): ko imamo rešen križ, se lotimo spodnjih dveh plasti. V vsakem kotu potrebujemo po eno kotno in eno robno kockico. Tako robna kot kotna kockica sta lahko na osmih različnih lokacijah, pri čemer ima robna kockica tri možne orientacije, kotna pa dve. Število možnih kombinacij za en kot kocke je tako $8 * 3 * 8 * 2 = 384$. Za praktično uporabo pa je dovolj, da se naučimo 41 algoritmov, če pri tem predpostavimo, da so ostali trije koti kocke že rešeni in da je rotacija zgornje plasti trivialna operacija.
   Izkaže se, da je za ta korak potrebno največ rotacij in posledično časa. Kljub temu, da nam zadostuje 41 algoritmov, pa se je tudi ta korak priporočljivo naučiti intuitivno.



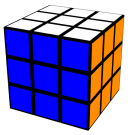
Slika 2.8: Prvi dve plasti

3. Orientacija zadnje ploskve (OLL - Orentation of Last Layer): v tem koraku rešimo zadnjo ploskev, tako da imajo vse kockice isto barvo, prstana pa ne upoštevamo. Za optimalno rešitev tega koraka se moramo naučiti 57 algoritmov, ki nam vsako možno postavitev kockic zarotirajo v ploskev z isto barvo. Pri počasnejši metodi celotno plast rešimo tako, da najprej sestavimo križ, nato pa pravilno orientiramo še preostale štiri kotne kockice. Pri tej verziji se moramo naučiti devet algoritmov.

Slika 2.9: Orientacija zadnje ploskve

4. Permutacija zadnje plasti (PLL - Permutation of the Last Layer): v

tem koraku rešimo še zadnjo plast, tako da dobimo rešeno kocko. Kockice med seboj premikamo tako, da spreminjamo le lokacijo, ne pa tudi orientacije. Najhitrejša je metoda z enaindvajsetimi algoritmi, ki nudijo rešitev za vse možne kombinacije kockic. Uporabimo pa lahko tudi počasnejšo metodo v dveh korakih, kjer se moramo naučiti šest algoritmov.



Slika 2.10: Rezultat permutacije zadnje plasti

2.2.2 Petrusova metoda

Metodo je iznašel Lars Petrus, ki se s hitrostnim reševanjem Rubikove kocke ukvarja že od samega začetka te discipline. V povprečju je za rešitev kocke s to metodo potrebno od 45 do 60 rotacij, primerna pa je tudi za začetnike. Prednost te metode je, da imamo po končanem prvem koraku na voljo tri plasti, ki jih lahko med seboj poljubno rotiramo, ne da bi pri tem uničili že končane manjše kocke. Sestoji iz sedmih korakov:

1. Sestavi kocko velikosti 2x2x2: prvi korak je en kot kocke, kjer v pravo kombinacijo sestavimo eno kotno, tri robne in tri sredinske kockice. Dobimo kocko s štirimi pravilno postavljenimi kockicami.

2. Povečaj rešeno kocko na velikost 2x2x3: pravilno postavljenim kockicam dodamo še eno kotno in dve robni kockici. S tem se število pravilno postavljenih kockic poveča na sedem.

3. Popravi orientacijo kotnih kockic: v tem koraku si pripravimo kotne kockice, tako da bodo pravilno orientirane.

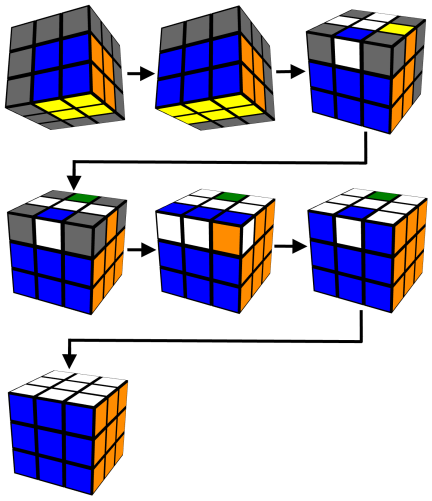
4. Reši dve celi ploskvi: najtežji korak metode. Če želimo ohraniti že rešene kockice, imamo na voljo le še dve plasti, ki ju lahko rotiramo.

Cilj je, da na pravilno mesto postavimo še dve dodatni kotni in tri robne kockice. Dobimo kocko z dvanajstimi pravilno postavljenimi kockicami.

5. Pravilno postavi kotne kockice na zadnji plasti: preostale štiri kotne kockice postavimo na pravo mesto, pri čemer ni nujno, da so tudi pravilno orientirane.

6. Orientiraj kotne kockice na zadnji plasti: kotne kockice, ki smo jih na pravo lokacijo postavili v prejšnjem koraku, orientiramo tako, da bodo vse barve na pravem mestu.

7. Pravilno postavi robne kockice na zadnji plasti: zadnji korak je razporeditev preostalih štirh robnih kockic na zadnji plasti. Rezultat je kocka, ki ima na vseh šestih ploskvah isto barvo.



Slika 2.11: Koraki pri Petrusovi metodi

2.3 Rubikova kocka v računalništvu

Zaradi svoje popularnosti je Rubikova kocka predmet številnih projektov v računalništvu. Aplikacij, kjer lahko uporabnik vnese svojo kocko in dobi algoritem za rešitev, je že precej, od namiznih, spletnih pa vse do mobilnih

različic (naša aplikacija se od ostalih sicer razlikuje v tem, da kocko reši z metodo Fridrich po posameznih korakih). Na voljo imamo tudi množico pomožnih programov, na primer za vodenje statistike porabljenega časa za rešitev, časomerov, generiranje kratkih algoritmov za premešanje kocke in podobno.

Ljubitelji po celem svetu pa razvijajo tudi namenske robote, ki znajo kocko fizično rešiti, seveda v povezavi z ustrezno programsko opremo. Takšna rešitev je zapleten problem, saj je potrebno razviti mehanizme za rotiranje plasti kocke, prepoznavo barv preko senzorjev (večinoma kamer, pa tudi preko telefonov) in upravljanje celotnega sistema (večinoma s pomočjo računalnika, novejše rešitve pa tudi zgolj s pomočjo telefonov). Takšni roboti že dosegajo čase pod eno sekundo, kar je za ljudi seveda nedosegljiv rezultat.

Računalniški algoritmi pa so igrali tudi pomembno vlogo pri iskanju zgornje meje rešitve, ki predstavlja največje število rotacij, potrebnih za rešitev vsake kocke. Dejstvo, da smo zgornjo mejo našli šele leta 2010 po dolgih letih naporov matematikov in programerjev, nam dokazuje, za kako zapleten problem gre. S pomočjo naprednih algoritmov in super zmogljivih računalnikov, ki jih je nudil Google, so raziskovalci za zgornjo mejo (imenovano tudi božje število) končno določili 20 rotacij.

Poglavje 3

Uporabljena orodja

Aplikacijo smo razvili v razvijalskem okolju Visual Studio 2010, napisana je v jeziku C# in teče na .NET ogrodju verzije 4.0 ali več, za podatkovno bazo pa uporablja Microsoft SQL Server 2008 R2. Grafični vmesnik je bil razvit s pomočjo tehnologije Windows Presentation Foundation (WPF), ki ponuja velik nabor gradnikov za delo z interaktivnimi objekti v trorazsežnem prostoru. Nepogrešljiv del razvoja je bilo testiranje funkcionalnosti naših objektov s pomočjo orodja nUnit (unit testing).
Za uporabo aplikacije potrebujemo .NET ogrodje verzije 4.0 ali več in podatkovni strežnik Microsoft SQL Server 2008. Podatkovni strežnik nam je bil v pomoč pri generiranju algoritmov za reševanje kocke in bi se lahko na enostaven način nadomestil z drugačnim načinom hranjenja podatkov, na primer s tekstovnimi ali XML datotekami. Odvisnosti od podatkovnega strežnika bi se s tem znebili, aplikacija pa bi tekla na vseh sistemih z nameščenim operacijskim sistemom Windows in .NET ogrodjem verzije 4.0.

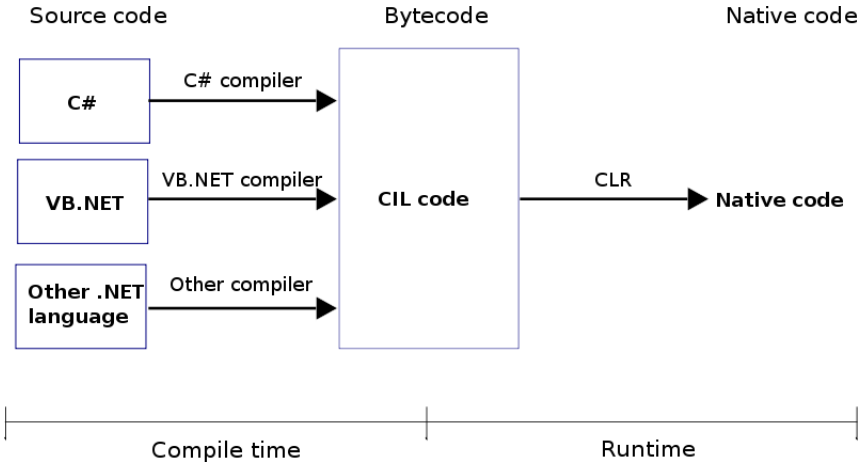
3.1 Microsoft .NET ogrodje

Microsoft .NET ogrodje teče na operacijskem sistemu Windows, v okrnjeni različici pa tudi na raznih Linux distribucijah, Androidu in ostalih sistemih. Gre za bogat ekosistem tehnologij v osrčju katerega je Common Language

Runtime, ki skrbi za pravilno zaganjanje aplikacij z vidika varnosti in upravljanja s spominom. Nekaj glavnih komponent .NET ogrodja:

- Common Intermediate Language (CIL): vse izvršilne datoteke v .NET ogrodju so prevedene v CIL jezik, definiran v Common Language Infrastructure specifikaciji (CLI). Je objektni programski jezik na zelo nizkem nivoju in je procesorsko in platformno neodvisen. Pri razvoju lahko sicer uporabimo različne programske jezike, na primer C#, Visual Basic .NET, C++/CLI, F#, IronPython, v fazi prevajanja pa se bodo vedno kreirale ustrezne CIL datoteke. Dobro je tudi vedeti, da lahko iz izvršilnih datotek dobimo kodo v višjenivojskih jezikih (na primer C# ali C++/CLI), večinoma z nekaj sintaktičnimi razlikami.

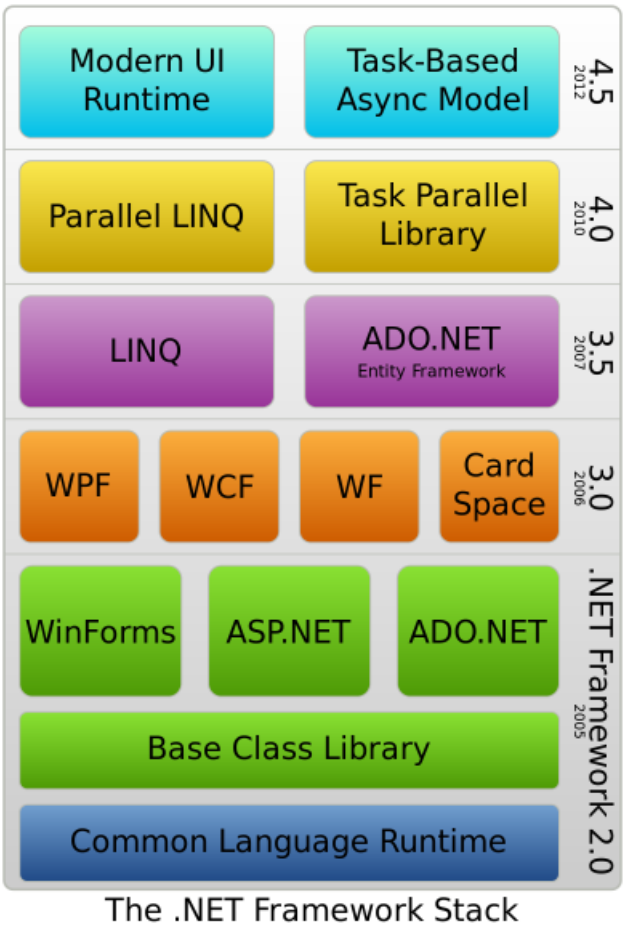
- Common Language Runtime (CLR): CLR zaganja izvršilne datoteke, pri čemer kodo v CLI jeziku sproti prevaja v strojno kodo in jo posreduje procesorju. Ta vmesni prevajalnik se imenuje Just in Time prevajalnik (JIT), ki vsak del kode ob prvem zagonu prevede v procesorju razumljiv jezik. V tem koraku se izvede tudi optimizacija glede na okolje, v katerem zaganjamo aplikacijo. Hkrati pa CLR skrbi tudi za pravilno delovanje aplikacij, na primer upravljanje s pomnilnikom (Garbage Collector - GC), upravljanje z napakami, upravljanje z nitmi,...



Slika 3.1: Common Language Runtime (CLR)

- Base Class Library (BCL): ogromna zbirka objektov in gradnikov, ki so na voljo razvijalcem za lažji in hitrejši razvoj aplikacij. Zbirka je razdeljena v imenske prostore (namespace), znotraj teh pa so objekti s podobno funkcionalnostjo. Primer pomembnejših imenskih prostorov: System (glavni tipi, matematične funkcije,...), System.Collections (množice, seznami, slovarji,...), System.IO (povezava do datotečnega sistema), System.Data, System.Linq,...

.NET ogrodje je zaživelo leta 2002 z verzijo 1.0 in se aktivno razvija, trenutna verzija je 4.5. V zadnjem času pridobiva na popularnosti tudi zaradi odprtosti platforme, kar omogoča posameznikom in podjetjem izdelavo dodatkov in namenskih aplikacij, na primer analitičnih orodij za pregled porabljenega pomnilnika in optimizacijo kode.



Slika 3.2: Glavne verzije .NET ogrodja

3.2 Microsoft Visual Studio

Je verjetno najbolj uporabljeno razvijalsko okolje v .NET ekosistemu, v naši diplomski nalogi smo uporabili verzijo 2010. Vsa orodja, ki so nujno potrebna za razvoj aplikacij, so že vključena v osnovno verzijo, to sta predvsem prevajalnik za več pogostejših jezikov (C#, Visual Basic .NET, C++/CLI) in razhroščevalnik. Vsebuje pa tudi orodja, ki programerju olajšajo delo, na primer napreden urejevalnik kode z IntelliSense tehnologijo, gradnike za izdelavo grafičnega vmesnika namiznih in spletnih aplikacij, možnost vizu-

alnega urejanja objektov in podatkovne sheme ter uporabo vtičnikov. Prav z vtičniki si lahko delo zelo olajšamo, naj omenim samo nuget, s katerim lahko v svoj projekt na enostaven način dodamo javno objavljene knjižnice iz spleta.

3.3 Programski jezik C#

C# je programski jezik, definiran v standardu ECMA-334. Zasnovan je bil predvsem kot objektni programski jezik, v zadnjem času pa dobiva primesi tudi ostalih paradigem. Razvijal se je hkrati z .NET ogrodjem, trenutno je pri verziji 5.0. Pomembnejše verzije sovpadajo z novimi različicami razvojnega orodja Visual Studio in .NET ogrodja, pri čemer pa si številke verzij ne sledijo v istem zaporedju.

Verzija C#	Verzija .NET	Verzija VS	Novosti
C# 1.0	.NET 1.0	VS2002	Osnovna verzija
C# 1.2	.NET 1.1	VS2003	Manjši popravki in dodatki
C# 2.0	.NET 2.0	VS2005	Generični tipi, anonimne metode, iteratorji
C# 3.0	.NET 3.5	VS2008	Lambda funkcije, anonimni tipi, var sintaksa
C# 4.0	.NET 4.0	VS2010	Dinamični tipi, opcijski parametri
C# 5.0	.NET 4.5	VS2012	Async in await ukaza

3.4 WPF

Naša aplikacija mora omogočati prikaz kocke v trodimenzionalnem prostoru, interakcijo s kocko preko miške in animacije posameznih rotacij. Zaradi teh

zahtev smo za grafični vmesnik izbrali tehnologijo Windows Presentation Foundation (WPF), ki nudi vsa potrebna orodja za naštete funkcionalnosti. WPF je vključen v .NET ogrodje od verzije 3 naprej in se pogosto uporablja kot alternativa običajnim WinForms namiznim aplikacijam. Za izgradnjo grafičnega vmesnika se uporablja jezik XAML (Extensible Application Markup Language), deklarativen jezik, ki temelji na XML sintaksi, omogoča pa tudi napredne tehnologije kot na primer povezavo javnih lastnosti objektov iz poslovne logike na objekte iz grafičnega dela (data binding), nastavljanje dogodkov (eventing) in prožilcev (triggering), animacije, efekte,... Vizualni del temelji na knjižici DirectX, ki zna kodo poganjati tudi na grafičnem procesorju. Grafično zahtevne aplikacije so zato v tehnologiji WPF načeloma bolj odzivne kot v Windows Forms. Pomembna lastnost pa je tudi neodvisnost od ločljivosti zaslona. Tako se na primer nek tekst ob povečanju ločljivosti ne pomanjša, ampak se prilagodi zaslonu, enako velja seveda za vse ostale gradnike grafičnega vmesnika.

3.5 SQL Server 2008 R2

Algoritme in pomožne podatkovne strukture smo si hranili v obliki enostavnih tabel s pomočjo podatkovnega strežnika Microsoft SQL Server. Microsoft SQL strežnik je sistem za upravljanje relacijskih zbirk podatkov, njegova glavna naloga je hramba podatkov in poizvedbe nad njimi, vsebuje pa tudi naprednejše tehnologije, na primer podporo za referenčno integriteto podatkov, shranjevanje pogosto dostopanih podatkov v pomnilniku, transakcije, nadzor nad sočasnim izvajanjem transakcij, zna pa tudi uporabljati metode, napisane v .NET ogrodju. Za poizvedbe se uporablja programski jezik Transact-SQL, ki je implementacija SQL-92 standarda.
Microsoft SQL Server pa nudi tudi širok nabor servisov za lažje upravljanje z bazami, na primer Analysis Services (uporablja se predvsem za podatkovno rudarjenje in kreiranje OLAP kock), Reporting Services (izdelava poročil, monitoring) in Integration Services (uvozi in izvozi podatkov, konverzije med

različnimi formati).
V diplomskem delu smo uporabili brezplačno verzijo Microsoft SQL Server 2008 R2 Express in Microsft SQL Server Management Studio za kreiranje baz in tabel, saj smo izkoristili le najosnovnejše funkcionalnosti - zapisovanje in branje v enostane tabele brez relacij med njimi.

3.6 nUnit

nUnit je orodje za izvajanje funkcionalnih testov (unit testing) nad javnimi metodami v naši kodi. Pri takšnih testih najpomembnejše metode testiramo tako, da jih zaženemo z različnimi vhodnimi prametri in nato preverimo, če se rezultat metode ujema s specifikacijami. Pri tem ni dovolj, da metodo preizkusimo zgolj za nek najpogostejši vhodni parameter, pač pa zavedno iščemo pogoje, kjer bi metoda lahko vrnila napačen rezultat (na primer če metoda pričakuje en celoštevilski parameter, moramo stestirati tudi vse robne pogoje, v .NET okolju je to Int32.MinValue in Int32.MaxValue, pa tudi z nekaj pravilnimi vhodnimi podatki, pri čemer točno vemo, kakšno vrednost mora metoda vrniti).
Funkcionalni testi so bili pri razvoju naše aplikacije izjemnega pomena, saj nam služi kot varovalna mreža, ki skrbi za to, da določena funkcija vrača isti rezultat ne glede na to, kako smo jo spreminjali tekom razvoja. Primer funkcionalnega testa, kjer preverimo stanje kocke ob rotaciji zgornje plasti v smeri urinega kazalca:

```
var cube = new Cube();

cube.Rotate("U");
Assert.That(cube.UpFace.ToString() == "WWWWWWWWW");
Assert.That(cube.FrontFace.ToString() == "OOOBBBBBB");
Assert.That(cube.RightFace.ToString() == "GGGOOOOOO");
Assert.That(cube.BackFace.ToString() == "RRRGGGGGG");
Assert.That(cube.LeftFace.ToString() == "BBBRRRRRR");
```

```
Assert.That(cube.DownFace.ToString() =="YYYYYYYYY");
```

Najpomembnejši test pa je kontrola stanja kocke po tem, ko smo jo rešili z našo metodo:

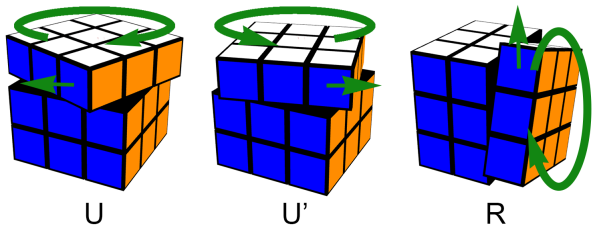
```
var cube = new Cube();
cube.Shuffle();
var solution = FridrichSolver.Solve(cube);
var cubeSolved = solution.GetLastState();
Assert.IsTrue(IsCubeSolved(cubeSolved));
```

Poglavje 4

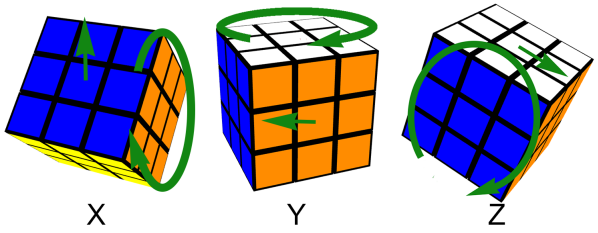
Implementacija

4.1 Opis aplikacije

Glavna funkcionalnost aplikacije je vizualni prikaz rotacij, ki so potrebne za rešitev vhodne kocke. Uporabnik lahko določi začetno stanje kocke, tako da za vse vidne plasti kockic določi eno izmed šestih standardnih barv. Aplikacija pri potrditvi vnosa preveri, ali je začetno stanje pravilno, torej da je s pravilno kombinacijo vseh možnih rotacij kocko možno spraviti v pravilno stanje. Uporabnik se namreč lahko pri vnosu zmoti, tako da na primer eno plast kockice pozabi obarvati ali pa jo obarva napačno.
Kocka na zaslonu je interaktivna, zato lahko uporabnik na njej izvede poljubno rotacijo ali kombinacijo rotacij. Interakcija s kocko je zelo enostavna in poteka s pomočjo računalniške miške. Posamezne plasti rotiramo na način 'primi in potegni', kjer določeno kockico primemo z levo miškino tipko in jo potegnemo v želeni smeri. Celotno kocko rotiramo na isti način, le da kocko primemo in potegnemo z obema miškinima tipkama.

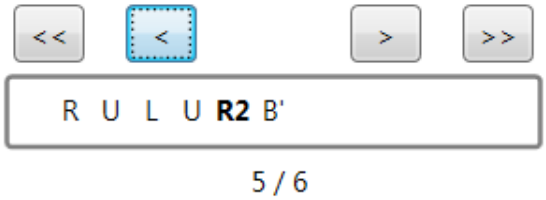


Slika 4.1: Primer rotacije plasti



Slika 4.2: Primer rotacije cele kocke

Pod kocko je prikazana zgodovina rotacij, med katerimi se lahko sprehajamo (na začetek, nazaj, naprej, na konec), uporabna pa je tudi v primeru, ko se pri rotaciji zmotimo in jo hočemo popraviti. Ob prehodu med rotacijami se izvede ustrezna animacija na kocki (rotacija celotne kocke ali pa posamezne plasti) in posodobi kontrola za prikaz števila vseh rotacij ter zaporedne številke trenutne rotacije.

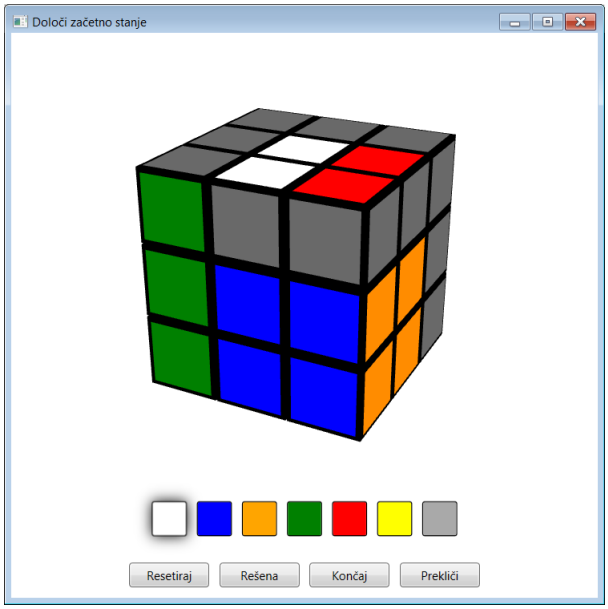


Slika 4.3: Kontrole

Po zagonu aplikacije se nam prikaže kocka v naključnem stanju, na voljo pa imamo več možnosti:

- Premešaj: Na končani kocki se izvede petdeset naključnih rotacij iz celotnega nabora rotacij (rotacija ene ali dveh ploskev, rotacija cele kocke), tako da dobimo naključno premešano kocko. Vsi ostali gradniki se ponastavijo na začetno stanje.

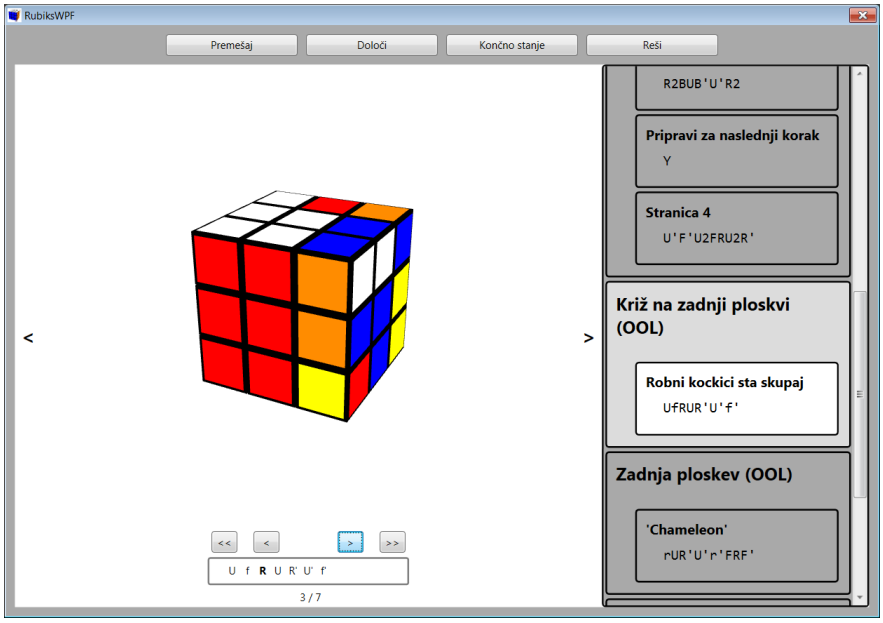
- Določi: uporabnik lahko določi začetno stanje, če želi na primer rešiti kocko, ki jo ima fizično v rokah, ali pa le vaditi posamezne algoritme in metode. Za vsako vidno plast kockic mora določiti eno izmed šestih standardnih barv (bela, modra, oranžna, zelena, rdeča ali rumena). Po zaključenem vnosu aplikacija preveri, če je vnešena kocka v pravilnem stanju. Izvedejo se kontrole števila barv, obstoja vseh kockic (na primer vedno moramo imeti eno robno kockico z modro in belo barvo, eno kotno kockico z belo, modro in oranžno barvo itd.). Prav tako se preverijo orientacije kockic. Določena stanja namreč niso dovoljena. Primer je stanje, kjer so vse kockice v pravilnem stanju, le ena robna ima napačno orientacijo. Za takšne primere je dokazano, da kocke ne moremo rešiti z nobeno možno kombinacijo rotacij.
  Na voljo imamo dva pomožna gumba, eden kocko resetira (nastavi vse kockice na nedefinirano barvo, razen sredinskih), drugi pa kocko postavi v rešeno stanje. S klikom na gumb Končaj vnešeno stanje potrdimo in se vrnemo v glavno aplikacijo.



Slika 4.4: Določi začetno stanje

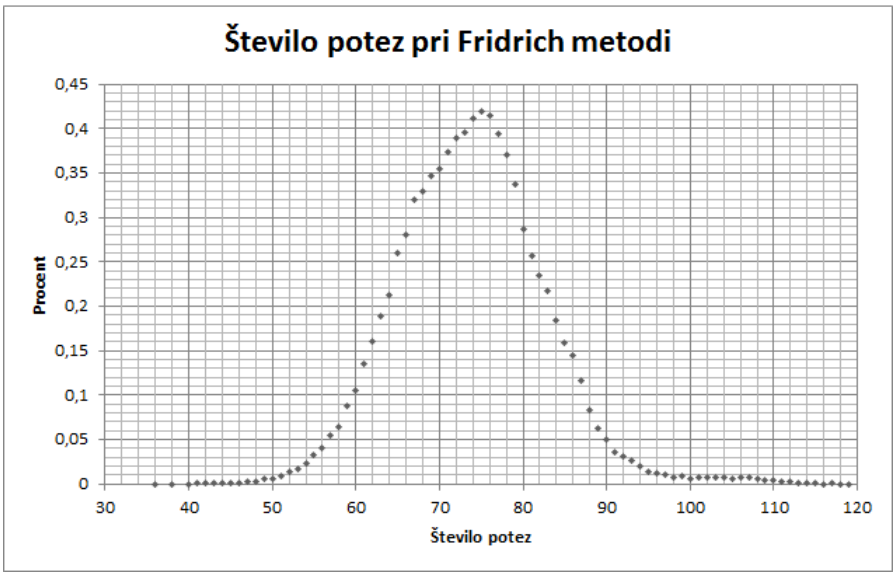
- Končno stanje: prikaže se kocka v končnem stanju, vsi ostali gradniki se postavijo na začetno stanje.

- Reši: v vsakem trenutku imamo na voljo možnost prikaza korakov za rešitev kocke. Pri tem se uporabi metoda Fridrich, ki proces reševanja razdeli na več korakov.
  Celotna rešitev je predstavljena v desnem delu aplikacije, kjer je vsak uporabljen algoritem poimenovan in opisan z zaporedjem rotacij. Posamezni algoritmi so združeni v pet korakov (križ, prvi dve plasti, orientacija zadnje ploskve, permutacija zadnje plasti). Manipulacija s kocko je v tem načinu omogočena do te mere, da omogoča le začasno rotiranje cele kocke v neki smeri, po izpuščeni miškini tipki pa se kocka zavrti nazaj. Na levi in desni strani kocke imamo na voljo dva gumba, s katerimi se premikamo med posameznimi algoritmi ali koraki rešitve.



Slika 4.5: Prikaz rešitve kocke

Naša metoda kocko reši s povprečno 73 potezami, kar je malo slabše od praktičnih ocen. Največ potez izgubimo pri kockah, ki jih je intuitivno mogoče rešiti z relativno malim številom rotacij, pa tudi pri predzadnjem koraku (orientacija zadnje ploskve), ki smo ga razdelili na dva koraka. Najdaljši algoritem je dolg 119 potez, najkrajši pa 36.



Slika 4.6: Povprečno število potez pri metodi Fridrich

4.2 Struktura kocke

V knjižnici imamo na voljo objekte, ki predstavljajo kocko (kocka, kockica, plast, ploskev) in metode, ki izvajajo rotacije nad kocko (rotiraj kocko po osi X, rotiraj zgornjo plast v smeri urinega kazalca itd.). Vsebuje tudi metode za reševanje kocke in pomožne metode za generiranje rešitev.
Ker je bil cilj diplomske naloge predstavitev Rubikove kocke v objektnem programskem jeziku, je struktura kocke implementirana kot nabor povezanih objektov, ki predstavljajo fizične dele kocke. Predvsem gre tukaj za posamezne kockice, ki so grupirane glede na svojo lokacijo (center, rob, kot). Dejstvo, da je vsaka kockica svoj razred, nam olajša delo pri grafičnem vmesniku, saj jo lahko tako na enostaven način prikažemo v trorazsežnem prostoru.
V nadaljevanju je predstavljenih nekaj pomembnejših razredov v aplikaciji.

4.2.1 CubicleBase

Abstrakten razred, iz katerega dedujejo vse manjše kockice. Vsebuje šest lastnosti, ki predstavljajo vsako ploskev kockice:

- U: up (zgoraj),
- F: front (spredaj),
- R: right (desno),
- B: back (zadaj),
- L: left (levo),
- D: down (spodaj.

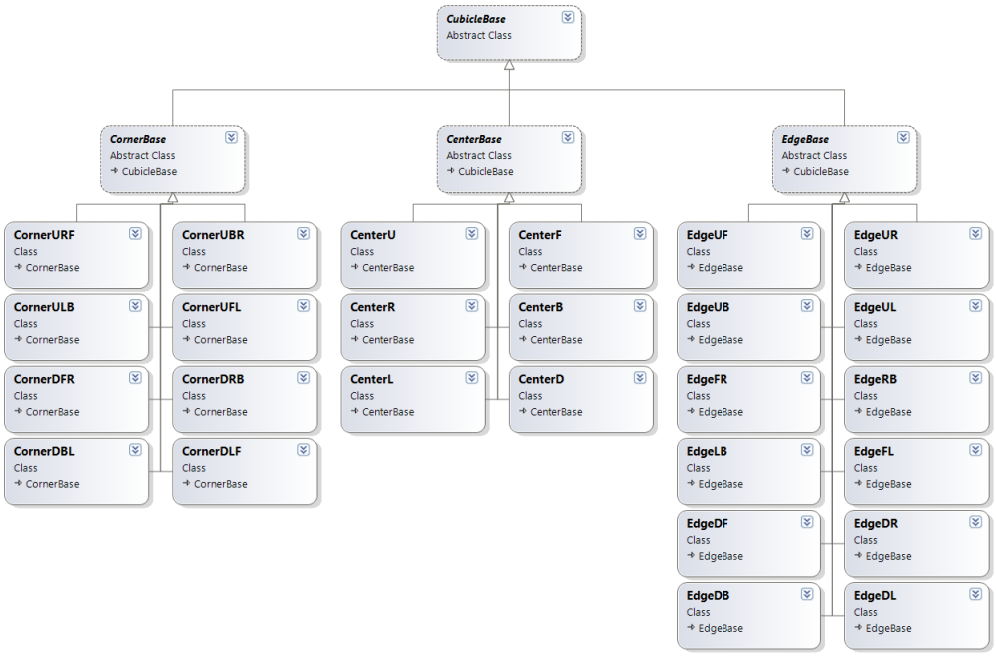
Vsaka ploskev je predstavljena s tipom char, ki je lahko iz nabora standardnih barv in ene pomožne barve:

- W: white (bela),
- B: blue (modra),
- O: orange (oranžna),
- G: green (zelena),
- R: red (rdeča),
- Y: yellow (rumena),
- H: hidden (skrita ploskev, ki se navzven ne vidi).

Razred CubicleBase ima tri podrazrede:

- CenterBase: abstrakten razred, ki združuje vse razrede, ki predstavljajo centralne kockice: CenterU (zgoraj), CenterF (spredaj), CenterR (desno), CenterB (zadaj), CenterL (levo) in CenterD (spodaj).
- EdgeBase: prav tako abstrakten razred, ki združuje vse razrede, ki predstavljajo robne kockice: EdgeUF, EdgeUR, EdgeUB, EdgeUL, EdgeFL, EdgeFR, EdgeRB, EdgeLB, EdgeDF, EdgeDR, EdgeDB, EdgeDL. Definiranih ima nekaj abstraktnih lastnosti (PrimaryFace in SecondaryFace, ki vsebujeta barvi obeh vidnih ploskev) in metod.

- CornerBase: abstrakten razred, ki združuje vse razrede, ki predstavljajo kotne kockice: CornerDBL, CornerDFR, CornerDLF, CornerDRB, CornerUBR, CornerUFL, CornerULB, CornerURF. Definiranih ima nekaj abstraktnih lastnosti (PrimaryFace, SecondaryFace in TertiaryFace, ki vsebujejo barve vseh treh vidnih ploskev) in metod, na primer ContainsColor (preveri, če kotna kockica vsebuje podano barvo) ter ContainsColors (preveri, če kotna kockica vsebuje vse tri podane barve).



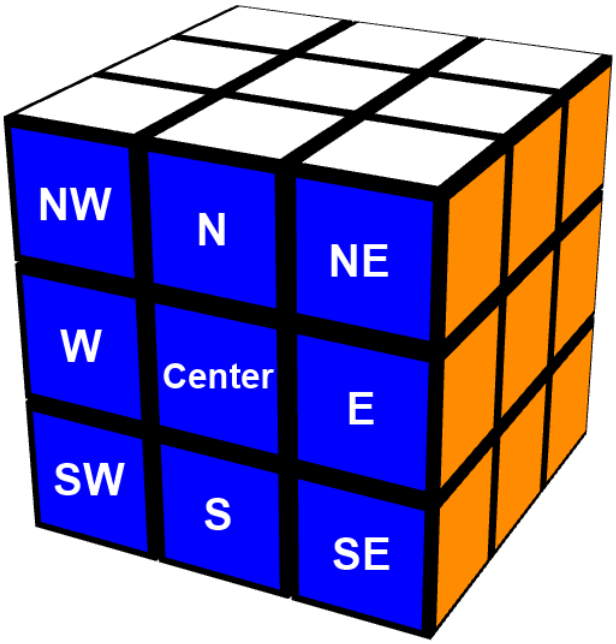
Slika 4.7: Razredni diagram strukture kockic

4.2.2 Face

razred predstavlja eno ploskev kocke in vsebuje devet lastnosti:

- NW: Northwest (severozahod),
- N: North (sever),

- NE: Northeast (severovzhod),
- W: West (zahod),
- Center: (center),
- E: East (vzhod),
- SW: Southwest (jugozahod),
- S: South (jug),
- SE: SouthEast (jugovzhod).



Slika 4.8: Prikaz poimenovanja kockic na eni ploskvi

4.2.3 Cubicles

Razred, ki vsebuje vse manjše kockice (6 centralnih, 12 robnih in 8 kotnih kockic) in vseh šest ploskev (zgoraj, spredaj, desno, zadaj, levo, spodaj).

Dejansko stanje kocke je shranjeno v manjših kockicah, ploskve pa so le pogledi na kocko z določene strani in nam stanja kocke ne spreminjajo. Primer definicije zgornje ploskve:

```
public Face UpFace
{
    get
    {
        return new Face(
            CornerULB.U, EdgeUB.U, CornerUBR.U,
            EdgeUL.U, CenterU.U, EdgeUR.U,
            CornerUFL.U, EdgeUF.U, CornerURF.U
        );
    }
}
```

Nekaj pomembnejših metod razreda Cubicles:

- GetCenterCubicles: metoda nam vrne seznam vseh centralnih kockic.
- GetEdgeCubicles: metoda nam vrne seznam vseh robnih kockic.
- GetEdgeCubicle(char primary, char secondary): glede na podani barvi poiščemo ustrezno robno kockico. Če kockica s takšnima barvama ne obstaja, vrnemo null.
- GetCornerCubicles: metoda nam vrne seznam vseh kotnih kockic.
- GetCornerCubicle(char primary, char secondary, char tertiary): glede na podane barve poiščemo ustrezno kotno kockico. Če kockica s takšnimi barvami ne obstaja, vrnemo null.
- Metode za rotacijo cele kocke po oseh X, Y ali Z: javne metode, ki znajo posamezne kockice obarvati tako, da ustrezajo končnemu stanju

podane rotacije. Za os X imamo zato na voljo tri metode: RotateAxisXClockWise (rotacija za 90 stopinj v smeri urinega kazalca), RotateAxisXAntiClockWise (rotacija za 90 stopinj v obratni smeri urinega kazalca) in RotateAxisXDouble (rotacija za 180 stopinj). Podobne metode imamo na voljo tudi za osi Y in Z.

- Metode za rotacijo posamezne plasti: vsako izmed šestih plasti lahko rotiramo v treh smereh, zgornjo plast lahko na primer rotiramo z metodami RotateUpFaceClockWise (rotacija za 90 stopinj v smeri urinega kazalca), RotateUpFaceAntiClockWise (rotacija za 90 stopinj v obratni smeri urinega kazalca) in RotateUpFaceDouble (rotacija za 180 stopinj). Poleg stranskih plasti pa lahko rotiramo tudi tri notranje: srednjo (pokončna med levo in desno plastjo), ekvatorsko (ležeča med spodnjo in zgornjo plastjo) in pokončno (pokončna med sprednjo in zadnjo plastjo). Primer kode za metodo RotateUpFaceClockWise():

```
public void RotateUpFaceClockWise()
{
    var tempEdge = EdgeUF;
    EdgeUF = new EdgeUF(EdgeUR.U, EdgeUR.R);
    EdgeUR = new EdgeUR(EdgeUB.U, EdgeUB.B);
    EdgeUB = new EdgeUB(EdgeUL.U, EdgeUL.L);
    EdgeUL = new EdgeUL(tempEdge.U, tempEdge.F);

    var tempCorner = CornerURF;
    CornerURF = new CornerURF(CornerUBR.U, CornerUBR.B, CornerUBR.R);
    CornerUBR = new CornerUBR(CornerULB.U, CornerULB.L, CornerULB.B);
    CornerULB = new CornerULB(CornerUFL.U, CornerUFL.F, CornerUFL.L);
    CornerUFL = new CornerUFL(tempCorner.U, tempCorner.R, tempCorner.F);
}
```

  Iz kode je razvidno, da je možnost za napako zelo velika (še posebej, ker je takšnih funkcij več), zato je bilo pri razvoju funkcionalno testiranje izjemnega pomena.

- Metode za rotacijo po dveh plasti skupaj: tudi tukaj imamo za vsako rotacijo na voljo tri smeri, pri čemer pa ne rotiramo le ene plasti, pač

pa po dve skupaj. Za dve zgornji plasti imamo tako na voljo metode RotateUp2ClockWise, RotateUp2AntiClockWise in RotateUp2Double. Na srečo tukaj ni potrebno pisati kode od začetka, pač pa lahko posamezno rotacijo dveh plasti simuliramo s predefiniranimi rotacijami. Primer kode za rotacijo zgornjih in spodnjih dveh plasti v smeri urinega kazalca:

```
public void RotateUp2ClockWise()
{
    RotateUpFaceClockWise();
    RotateEquatorFaceAntiClockWise();
}

public void RotateDown2ClockWise()
{
    RotateDownFaceClockWise();
    RotateEquatorFaceClockWise();
}
```

Iz kode je razvidno, da moramo paziti na smer rotacije, saj je odvisna od smeri opazovanja.

4.2.4 Move

Predstavlja posamezno rotacijo kocke ali plasti. Poleg same rotacije se hrani tudi obratna rotacija, na primer za rotacijo spodnje plasti za 90 stopinj v smeri urinega kazalca imamo na voljo obratno rotacijo spodnje plasti za 90 stopinj v nasprotni smeri urinega kazalca. Vse rotacije so poimenovane, imena pa so dosegljiva preko lastnosti RotateNotation in UnRotateNotation. Rotacije se vršijo preko metod Rotate() in UnRotate().

4.2.5 Moves

Razred Moves se uporablja za shranjevanje zgodovine rotacij in prehod med njimi. Glavne metode so StepBackwards (skoči na prejšnji korak), StepForward (skoči na naslednji korak), JumpToBeginning (skoči na začetek) in JumpToEnd (skoči na konec). Rotacije so shranjene v povezanem seznamu objektov tipa Move, ki se posodablja hkrati s spremembami stanja kocke.
Poleg glavnih metod za prehode med koraki pa razred Moves vsebuje še metode AddAndRotate (dodaj rotacijo v zgodovino in jo izvedi) in ApplyAlgorithm (razbij algoritem na posamezne rotacije, le te pa dodaj v zgodovino in jih izvedi), Shuffle (na kocki izvedemo 50 naključnih rotacij) ter nekaj pomožnih metod.

4.2.6 Cube

Glavni razred aplikacije, ki predstavlja fizično kocko. Vsebuje vse metode za manipulacijo s kocko, pri čemer dejanske akcije delegira bolj specifičnim objektom, opisanih v prejšnjih točkah. Sem spadajo metode za rotacijo, undo/redo funkcionalnost, ter iskanje kockic glede na vhodne parametre.
Glavne lastnosti razreda Cube so objekti iz prej opisanih razredov (Cubicles, Face za vsako plast posebej in Moves). Poleg tega pa so na voljo še pomožne lastnosti kot na primer barva posamezne plasti (UColor, FColor, RColor, BColor, LColor in DColor, vse tipa char) in Algorithm (povezan seznam rotacij).
Za obveščanje o spremembah stanja kocke imamo na voljo dva dogodka:

- OnCubeRotated: proži se za vsako rotacijo posebej, zato da lahko grafični vmesnik prikaže ustrezno animacijo.

- OnCubeChanged: pri določenih akcijah ne želimo, da se prikaže vsaka rotacija, pač pa le zadnje stanje kocke. Tipičen primer je metoda Shuffle, ki naključno premeša kocko. V teh primerih se izvedejo ustrezne rotacije na kocki, OnCubeRotated dogodek pa se ne proži. Ob

koncu rotacij sprožimo OnCubeChanged dogodek, na podlagi katerega grafični vmesnik prikaže osveženo stanje kocke.

4.3 Solution

Rešitev kocke je predstavljena z razredom Solution, ki posamezne korake združi v skupine in jih poimenuje. Skupine so lahko v dveh nivojih zato da lahko rešitev prikažemo v enostavno razumljivi obliki. Primer naključne rešitve kocke:

Skupina/Podskupina	Algoritem
Križ	
Z enim algoritmom	ULFRB'D'F
Prvi dve plasti (F2L)	
Stranica	R'URF'UFU2RUR'
Pripravi za naslednji korak	Y
Stranica	B2L'B2LUURU2R2FRF'
Pripravi za naslednji korak	Y
Stranica	U2URU2R2FRF'
Pripravi za naslednji korak	Y
Stranica	U2R2UR2UR2U2R2
Križ na zadnji ploskvi (OLL)	
Robni kockici sta narazen	UFRUR'U'F'
Zadnja ploskev (OLL)	
Anti Sune	R'U'RU'R'U2R
Permutacija zadnje plasti (PLL)	
Permutacija U : a	R2U'R'U'RURURU'R
Permutacija zadnje plasti (PLL)	
Permutacija A : a	X'R'DR'U2RD'R'U2R2X

Kot je razvidno iz primera, se da rešitev kocke z metodo Fridrich lepo prikazati z dvonivojsko strukturo. Izkaže se, da se da na takšen način prikazati

tudi rešitve ostalih metod. Pri razvoju to pomeni, da je potrebno le implementirati novo metodo, vsa potrebna infrastruktura pa bo ostala na voljo (struktura kocke, struktura rešitve, grafični vmesnik).

Primer (poenostavljene) kode, ki služi za predstavitev rešitve kocke:

```
public class Step
{
    public readonly string Note;
    public readonly Cube Cube;
    public readonly Alg Algorithm;

    public Step(string note, Cube cube, Alg algorithm)
    {
        Note = note;
        Cube = cube;
        Algorithm = algorithm;
    }
}

public class GroupOfSteps : LinkedList<Step>
{
    public readonly string Note;

    public GroupOfSteps(string note)
    {
        Note = note;
    }
}

public class Solution : LinkedList<GroupOfSteps>
{
```

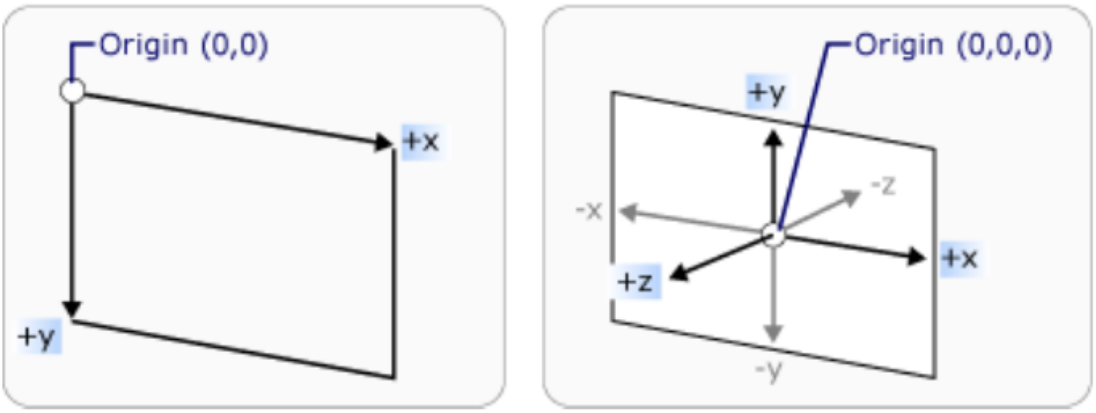
```
    private readonly Cube _initialInitialCube;

    public Solution(Cube initialCube)
    {
        _initialInitialCube = initialCube;
    }
}
```

Vsak korak rešitve (Step) vsebuje lastno instanco razreda Cube. To je pomembno za grafični vmesnik, kjer lahko za vsak posamezen korak vidimo kocko in prehajamo med rotacijami znotraj tega koraka.

4.4 Implementacija grafičnega vmesnika

WPF tehnologija privzeto nudi širok nabor objektov za delo v 3D okolju. Vse objekte lahko kreiramo in manipuliramo iz kode ali pa deklarativno v XAML datotekah. Za prikaz nekega objekta si najprej zgradimo graf objektov, sistem pa ga zna prikazati in osveževati po potrebi. Na voljo imamo napredne funkcije, na primer hit testing (iskanje objekta, ki ga je uporabnik izbral z miško), dogodke (podobno kot pri vseh ostalih objektih v .NET ogrodju, predvsem nas tukaj zanima interakcija z miško) in animacije.

Pogoj za delo s 3D objekti je poznavanje koordinatnega sistema, ki je definiran v 3D okolju. V primerjavi z 2D okoljem je Y os namreč usmerjena navzgor in ne navzdol. Z os pa je usmerjena proti opazovalcu.



Slika 4.9: Koordinatni sistem v 2D in 3D okolju

4.4.1 3D gradniki v WPF tehnologiji

Gradniki, ki smo jih uporabili pri diplomski nalogi:

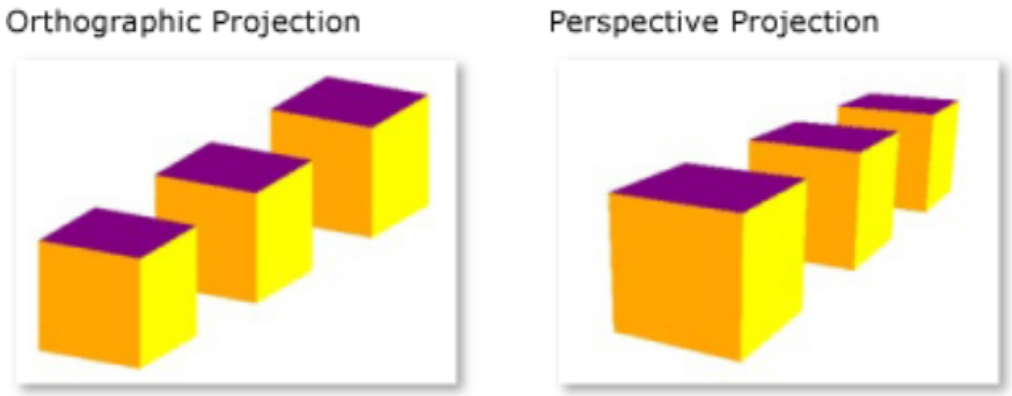
- Viewport3D: če želimo prikazati nek objekt ali skupino objektov v 3D okolju, moramo uporabiti Viewport3D gradnik, ki ga vstavimo v navaden dvodimenzionalni vizualni objekt, na primer Window ali Grid. Da pa lahko naš objekt vidimo, moramo uporabiti še kamero in osvetljavo.

- Camera: prav tako kot v resničnem svetu je tudi v WPF tehnologiji izjemno pomembno, kam je kamera postavljena in kam gleda. Če kamera ne gleda proti našim objektom, jih na zaslonu namreč ne bomo videli. Zato moramo kameri nastaviti vsaj ti dve osnovni lastnosti:

  - Position: pozicija kamere v koordinatnem prostoru, določena s točkami X, Y in Z,
  - LookDirection: vektor, kamor je kamera usmerjena.

  Ko imamo naš objekt na zaslonu, pa lahko kameri nastavljamo še ostale parametre, na primer UpDirection (vektor, ki nam pove orientacijo kamere, s katero gledamo proti objektom) in FieldOfView (vidni kot kamere).

WPF nudi dva tipa kamere - OrthographicCamera in PerspectiveCamera. V naši aplikaciji smo uporabili slednjo, saj je bližje realnemu svetu, kjer so objekti v bližini večji kot pa oddaljeni objekti.



Slika 4.10: Razlika med ortogonalno in perspektivno kamero

- Osvetljava: pri osvetljavi imamo na izbiro več možnosti:
  - DirectionalLight: osvetljava iz točno določene točke in smeri, pri čemer je izvor svetlobe zelo daleč,
  - PointLight: podobno kot DirectionalLight, le da je izvor svetlobe bližje,
  - SpotLight: svetloba prihaja iz neke točke v obliki stožca,
  - AmbientLight: vse površine so enakomerno osvetljene.

  V naši aplikaciji smo uporabili AmbientLight, saj so tako vse barve na kocki dobro vidne.

- GeometryModel3D: oblika vidnih objektov je definirana z njihovo geometrijo in materialom.
  Obstaja več tipov materialov, na primer DiffuseMaterial (svetlobo razpršijo po celi ploskvi), SpecularMaterial (svetleči materiali, kjer se svetloba odbija glede na vpadni kot) in EmissiveMaterial (materiali, ki svetlobo

oddajajo). Vidne ploskve kockic uporabljajo DiffuseMaterial, zato so vse barve enako dobro vidne.
Geometrija je določena s trikotniki, vsebovanimi v MeshGeometry3D objektu. Trikotnikom je potrebno določiti vsaj lastnosti Positions (točke vseh trikotnikov) in TriangleIndices (definira, katere točke so povezane v trikotnike). Pri povezovanju točk v trikotnike je potrebno paziti na orientacijo, saj je od tega odvisno ali gledamo na trikotnik od sprednje ali zadnje strani.

- ModelVisual3D: nudi metode in lastnosti, ki so skupne vsem vizualnim objektom, na primer hit testing in animacije. Dejanska vsebina, ki bo prikazana, se določi v lastnosti Children. Vse kockice iz naše aplikacije dedujejo od ModelVisual3D razreda, saj si s tem delo precej olajšamo, predvsem pri animacijah.

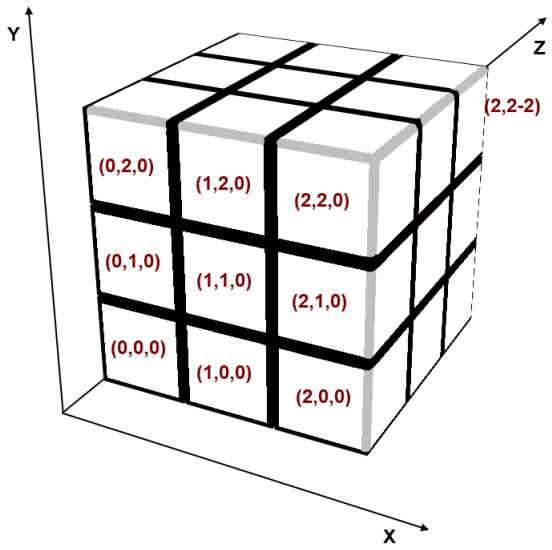
- Animacije: animacije zaganjamo preko Transform lastnosti na ModelVisual3D objektih. Najpogosteje objekte premikamo, obračamo in jim spreminjamo velikost, možno pa je tudi animiranje ostalih lastnosti, ki ustrezajo določenim pravilom (lastnost mora biti 'Dependency property', razred mora dedovati od DependencyObject razreda in implementirati IAnimatible interface). Razvijalcem ni potrebno skrbeti za poganjanje merilca časa, proženje dogodkov in osveževanja na grafičnem vmesniku, saj za to skrbi sistem sam.

4.4.2 Prikaz kocke

Tako kot sama struktura kocke je tudi grafični vmesnik implementiran objektno usmerjeno. To pomeni, da je vsaka kockica svoj razred, ki deduje od ModelVisual3D razreda. Vsaki kockici smo določili lokacijo v navideznem koordinatnem sistemu, zato jih lahko izrišemo z isto kodo s tem da upoštevamo odmik glede na njeno pozicijo.

Kockice imajo vseh šest ploskev obarvano črno, tudi tiste, ki se ne vidijo.

Za prikaz barvnih nalepkic smo dodali manjše ploskve na vidne površine in jih obarvali z ustrezno barvo.



Slika 4.11: Koordinatni sistem za kockice

Glavni razredi, ki nam omogočajo prikaz kocke:

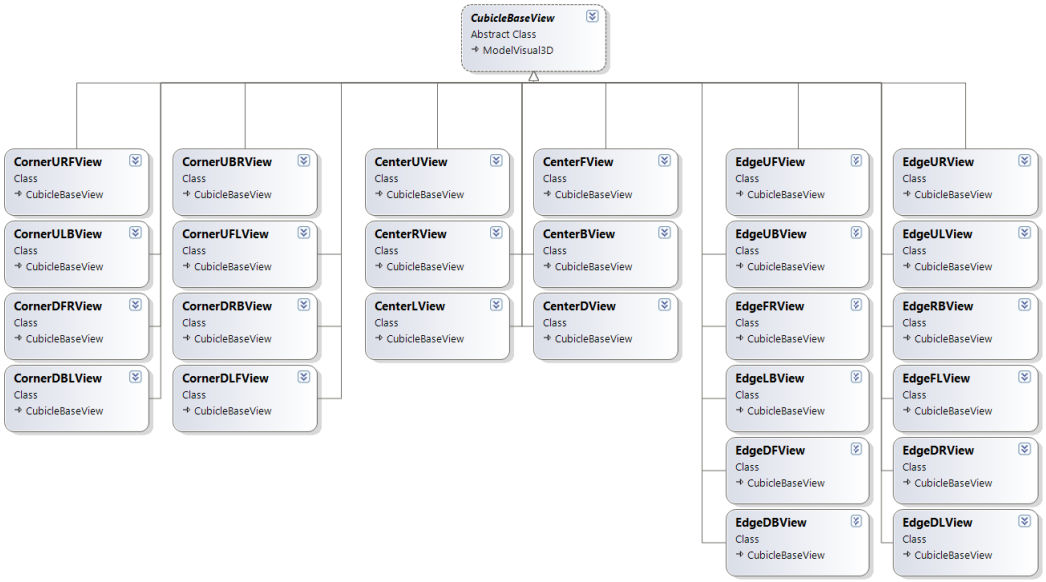
- CubeFaceVisual: predstavlja eno ploskev kockice, lahko je črna ali barvna. Deduje od razreda ModelVisual3D, zato ji moramo določiti le vsebino (lastnost Content), ki je tipa GeometryModel3D in pozicijo. GeometryModel3D je zgrajen iz MeshGeometry3D z ustreznimi točkami in povezavami med njimi in materialom DiffuseMaterial ustrezne barve.

- CubicleBaseView: abstraktni razred, iz katerega dedujejo vsi ostali razredi, ki predstavljajo kockice. Pomembnejše metode:

  - SetColor: nastavi barvo določeni ploskvi kockice.
  - GetMoveNotation: na podlagi ploskve in smeri potega vrnemo rotacijo, ki jo je potrebno izvesti. Na primer če kliknemo zgornjo ploskev spredje zgornje kockice in jo potegnemo v desno, je

potrebno izvesti rotacijo F (obrat sprednje plasti za 90 stopinj v smeri urinega kazalca). Če isto ploskev potegnemo v levo, bomo izvedli rotacijo F' (obrat sprednje plasti za 90 stopinj v nasprotni smeri urinega kazalca).

– GetBothMouseButtonsPressedMoveNotation: podobno kot GetMoveRotation, le da rotiramo celo kocko namesto posameznih plasti.



Slika 4.12: Razredni diagram kockic

- CubeView: razred, ki zna kocko prikazati na zaslonu. Ker deduje od razreda ModelVisual3D, je potrebno lastnosti Children le dodati posamezne kockice, za ostalo pa poskrbi sistem sam. CubicleView razred vsebuje tudi metode za animacijo rotacij.

  Seznam pomembnejših metod:

  – RenderCube: osveži kocko na zaslonu. Primer kode, kjer prikažemo le centralne kockice, podobno pa je za vse ostale:

```
Children.Add(new CenterUView(Cube.Cubicles.CenterU));
Children.Add(new CenterFView(Cube.Cubicles.CenterF));
Children.Add(new CenterRView(Cube.Cubicles.CenterR));
Children.Add(new CenterBView(Cube.Cubicles.CenterB));
Children.Add(new CenterLView(Cube.Cubicles.CenterL));
Children.Add(new CenterDView(Cube.Cubicles.CenterD));
```

  - Rotate: na podlagi podane rotacije se prikaže ustrezna animacija, ampak le na kockicah, ki v rotaciji nastopajo.
  - ApplyRotation: ko uporabnik klikne na kockico in jo potegne v neki smeri, aplikacija sproti prikazuje pravilno pozicijo rotirane plasti. To izvaja ravno metoda ApplyRotation, ki prejme kot parameter rotacijo in procent, za koliko je plast rotirana.
  - EndRotation: metoda se izvede, ko uporabnik spusti miškino tipko (MouseUp dogodek). Plast ali kocka se nato animira tako, da se postavi v končni položaj.

- UsrMoves: gradnik, ki vsebuje gumbe za prehod med koraki algoritma (naprej, nazaj, na začetek, na konec), trenutni algoritem in skupno število korakov.

- UsrCube: gradnik za prikaz kocke. Glavni element je Viewport3D, v katerem je prikazana kocka. Ta vsebuje kamero, osvetljavo in vsebnik _cubeContainer, ki mu programsko določimo vsebino. Poenostavljena XAML koda:

```
<UserControl>
    <Grid>
        <Viewport3D>
            <Viewport3D.Camera>
                <PerspectiveCamera Position="5,5,10"
                                   LookDirection="-5,-5,-10"
                                   UpDirection="0,1,0"
```

```
                                    FieldOfView="45" />
            </Viewport3D.Camera>

            <ModelVisual3D Content="{StaticResource Light}">
            </ModelVisual3D>

            <ModelVisual3D>
                <ModelVisual3D x:Name="_cubeContainer" />
            </ModelVisual3D>
        </Viewport3D>

        <CubeView:ViewportCanvas Background="Transparent">
        </CubeView:ViewportCanvas>

        <CubeView:UsrMoves ></CubeView:UsrMoves>
    </Grid>
</UserControl>
```

Kocko prikažemo tako, da vsebniku _cubeContainer v lastnost Children dodamo prej opisan objekt tipa CubeView.

Del UsrCube gradnika je tudi tako imenovan hit testing, s katerim preverimo, kateri objekt je uporabnik izbral. Proži se na MouseDown dogodek in izgleda tako:

```
VisualTreeHelper.HitTest(
    _viewport,
    null,
    myresult =>
    {
        Visual3D visualHit = ((RayHitTestResult)myresult).VisualHit;
        if (visualHit == null) return HitTestResultBehavior.Stop;
```

```
            var hit = visualHit as CubicleFaceVisual;
            if (hit != null)
            {
                _cubeView.SelectCubicleFace(hit);
                return HitTestResultBehavior.Stop;
            }

            // Don't go further...
            return HitTestResultBehavior.Stop;
        },
        new PointHitTestParameters(e.StartPoint)
    );
```

- ViewportCanvas: ViewportCanvas je prozorno platno, ki je prevlečeno čez kocko in je del gradnika UsrCube. Uporabljamo ga za lovljenje miškinih dogodkov, ki jih proži naprej. Pri ročni rotaciji, kjer uporabnik povleče del kocke, pa zna izračunati procent trenutne rotacije, da jo lahko objekt tipa CubeView prikaže.

4.4.3 Animacije

Rotacija je predstavljena z razredom AxisAngleRotation3D, ki mu določimo os vrtenja v obliki vektorja. Trajanje animacije je privzeto 0,5 sekunde. Ker gre za obrat, je začetna vrednost vedno 0, končna pa 90 ali 180, odvisno ali gre za enojni ali dvojni obrat.

Postopek za prikaz animacije:

- Najprej si pripravimo objekt tipa AxisAngleRotation3D, ki je odvisen od podane rotacije. Za rotacije X, R in L' imamo tako na primer rotacijo po osi, podani z vektorjem (-1, 0, 0), za rotacije X', R' in L pa z vektorjem (1, 0, 0).

- Nato določimo tip animacije, ki je v našem primeru DoubleAnimation s parametri From (0), To (90 ali 180) in Duration (0,5 sekunde).
- Vsem kockicam, ki v rotaciji nastopajo, nastavimo lastnost Transform na nov objekt tipa RotateTransform3D s parametrom AxisAngleRotation3D.
- Zaženemo animacijo.
- Ko se animacija zaključi, kocko izbrišemo in jo ponovno prikažemo. S tem se interno stanje kocke in njena grafična predstavitev sinhronizirata.

Primer poenostavljene kode za prikaz animacije:

```
var rotateData = GetrotateData(rotateNotation, from, to, duration);
var t = new RotateTransform3D(rotateData.Rotation);
foreach (var cubicle in GetCubicles(rotateNotation))
{
    cubicle.Transform = t;
}

((DoubleAnimation)rotateData.Animation).Completed +=
    (sender, e1) =>
    {
        container.Children.Clear();
        container.Children.Add(new CubeView(Cube));
    };

rotateData.Rotation.BeginAnimation(
    AxisAngleRotation3D.AngleProperty,
    rotateData.Animation
);
```

4.5 Generiranje pomožnih algoritmov za rešitev

Naša rešitev kocke temelji na prej pripravljenih algoritmih. Ko iščemo rešitev za podano kocko, moramo tako pri vsakem koraku pogledati kje so kockice, ki jih želimo postaviti na pravo mesto, poiskati v bazi ustrezen algoritem in ga zagnati. To poglavje opisuje, kako smo si algoritme pripravili.

4.5.1 Križ

Prvi korak je postavitev križa na spodnji plasti, kjer moramo vse štiri robne kockice postaviti na pravo mesto. Razmišljamo tako: na voljo imamo 12 robnih kockic, ki jih moramo postaviti na 4 pozicije, število vseh možnih kombinacij je torej $12 * 11 * 10 * 9$. Upoštevati moramo še orientacijo, kjer ima vsaka robna kockica dve možnosti, zato se število kombinacij poveča za $2^4 = 16$, skupaj na 190.080.

Seznam vseh 190.080 možnih kombinacij smo si pripravili s skripto v jeziku T-SQL s pomočjo cross join operacije (kartezični produkt):

```
with AllPositions as (
    select Id, Position
    from (
        values
            ((1), ('UF')), ((1), ('FU')),
            ((2), ('UR')), ((2), ('RU')),
            ((3), ('UB')), ((3), ('BU')),
            ((4), ('UL')), ((4), ('LU')),
            ((5), ('FR')), ((5), ('RF')),
            ((6), ('RB')), ((6), ('BR')),
            ((7), ('LB')), ((7), ('BL')),
            ((8), ('FL')), ((8), ('LF')),
            ((9), ('DF')), ((9), ('FD')),
            ((10), ('DR')), ((10), ('RD')),
```

```
            ((11), ('DB')), ((11), ('BD')),
            ((12), ('DL')), ((12), ('LD'))
    ) x(Id, Position)
)
select t1.Position, t2.Position, t3.Position, t4.Position
from AllPositions t1 cross join AllPositions t2
    cross join AllPositions t3
    cross join AllPositions t4
where
    t1.Id <> t2.Id and t1.Id <> t3.Id and t1.Id <> t4.Id
    and t2.Id <> t3.Id and t2.Id <> t4.Id
    and t3.Id <> t4.Id
```

Rezultat smo si shranili v tabelo, ki poleg štirih stolpcev za kockice vsebuje še stolpec za algoritem in pomožni stolpec 'Solved', ki pomeni ali imamo za določeno kombinacijo že rešitev (algoritem). Naslednji korak je bilo računanje algoritmov, ki je potekalo po tem postopku:

- Izmed vseh možnih kombinacij je točno ena takšna, ki ima križ že rešen. Tej kombinaciji smo stolpec 'Solved' nastavili na true.
- Nato v zanki zaganjamo naslednji postopek, dokler nimamo rešitve za vse pozicije:
  - Gremo skozi vse nerešene kombinacije.
  - Pri vsaki nerešeni kombinaciji izvedemo po eno izmed vseh možnih rotacij (U, U', U2, F, F', F2, R, R', R2, B, B', B2, L, L', L2, D, D', D2).
  - Če z eno izmed rotacij pridemo do kombinacije, za katero že imamo izračunan algoritem, potem shranjen algoritem in ustrezno rotacijo shranimo v bazo in nastavimo kombinaciji stolpec 'Solved' na 1.

Na takšen način smo si pripravili optimalne algoritme za vse možne kombinacije, pri čemer je najdaljši dolg 8 rotacij, povprečje pa je 5,8 rotacij:

Dolžina algoritma	Število
0	1
1	15
2	158
3	1394
4	9809
5	46381
6	97254
7	34966
8	102

4.5.2 Prvi dve plasti

V tem koraku je potrebno pravilno postaviti sprednjo robno in spodnjo kockico, tako da se bosta ujemali z vsemi ostalimi v njuni okolici. Ker je križ na spodnji strani že rešen, je lahko robna kockica na osmih lokacijah in v dveh orientacijah, kar nam da 16 možnih kombinacij. Kotna kockica je lahko prav tako na osmih možnih lokacijah, le da je vseh možnih orientacij 3. Za kotne kockice dobimo tako 24 možnih kombinacij, vključno z robnimi imamo zato skupaj 384 kombinacij. Tudi pri tem koraku smo si najprej pripravili vse možne kombinacije in jih shranili v SQL bazo, nato pa smo jim izračunali algoritme. Postopek izračuna algoritmov:

- Za reševanje prvih dveh plasti po metodi Fridrich imamo na voljo 41 standardnih algoritmov za najpogostejše kombinacije. Te algoritme smo uvozili za bazo, ostalo nam je 343 kombinacij brez algoritmov. Nato smo uporabili podoben postopek kot pri generiranju algoritmov za križ:

- Gremo skozi vse nerešene kombinacije.
- Pri vsaki kombinaciji zaženemo eno izmed vseh možnih rotacij in pogledamo, če imamo za novo kombinacijo že rešitev.
- Če dobimo rešitev, potem jo shranimo v bazo.
- Če rešitve ne dobimo, potem zaženemo po dve rotaciji skupaj.
- Če rešitve še vedno ne dobimo, zaženemo po tri rotacije skupaj.
- Postopek ponavljamo tako dolgo, da dobimo vse rešitve.

Tabela dolžin algoritmov za prvi dve plasti:

Dolžina algoritma	Število
0	1
3	2
4	26
5	21
6	32
7	86
8	169
9	47

4.5.3 Orientacija zadnje ploskve

Ta korak smo razdelili na dva dela:

1. Križ na zadnji ploskvi: na pravilno mesto moramo postaviti štiri kockice, ki so lahko na štirih možnih lokacijah, vsaka je lahko v eni izmed dveh orientacij. Skupno število kombinacij je tako 8, pri čemer je ena od teh kombinacij že rešena. Zato potrebujemo le 7 algoritmov, ki so dobro znani. Te algoritme smo si shranili v slovar, kjer je ključ vzorec kockic, vrednost pa ustrezen algoritem.

2. Celotna zadnja ploskev: Tudi tukaj imamo na voljo sedem algoritmov, ki smo jih dodali kar v kodo:

```
var algorithms = new Dictionary<string, string>
{
    {"1222", "R'U'RU'R'U2R"},
    {"3331", "RUR'URU2R'"},
    {"3232", "FRUR'U'RUR'U'RUR'U'F'"},
    {"2233", "RU2R2U'R2U'R2U2R"},
    {"1132", "R2DR'U2RD'R'U2R'"},
    {"3112", "rUR'U'r'FRF'"},
    {"2131", "F'rUR'U'r'FR"}
};
```

Ključ v slovarju je vzorec kotnih kockic, vrednost pa ustrezen algoritem.

4.5.4 Permutacija zadnje plasti

Pri permutaciji zadnje plasti smo uporabili 21 standardnih algoritmov, ki se uporabljajo pri metodi Fridrich. Podobno kot pri pripravi algoritmov za orientacijo zadnje ploskve smo si tudi tukaj vse algoritme shranili v slovar, kjer je ključ vzorec na kocki, vrednost pa ustrezen algoritem.
Vseh možnih kombinacij je sicer več kot dvajset, zato si je včasih potrebno kocko prej pripraviti; to storimo z ustreznimi rotacijami zgornje plasti ali pa celotne kocke po Y osi. Primer pomožnih rotacij, s katerimi si kocko pripravimo v stanje, ki je rešljivo z enim od 21 algoritmov:

```
private static IEnumerable<string> GetPLLRotations()
{
    yield return null;
    yield return "Y";
```

```
    yield return "Y'";
    yield return "Y2";
    yield return "U";
    yield return "U'";
    yield return "U2";
    yield return "UY";
    yield return "UY'";
    yield return "UY2";
    yield return "U'Y";
    yield return "U'Y'";
    yield return "U'Y2";
    yield return "U2Y";
    yield return "U2Y'";
    yield return "U2Y2";
}
```

4.6 Rešitev kocke

Rešitev kocke je ob prej pripravljenih algoritmih trivialna, vse kar moramo storiti je, da v trenutnem stanju preverimo kombinacijo kockic, ki jih želimo postaviti na pravo mesto in na podlagi tega poiskati ustrezen algoritem. Razred za rešitev kocke po metodi Fridrich:

```
public static class FridrichSolver
{
    public static Solution Solve(Cube cube)
    {
        var s = new Solution(cube);

        var stCross = SolverCross.Solve(s.LastCube);
        if (stCross != null) s.AddLast(stCross);
```

```
        var stFFL = SolverF2L.Solve(s.LastCube);
        if (stFFL != null) s.AddLast(stFFL);

        var stOOLCross = SolverOOLCross.Solve(s.LastCube);
        if (stOOLCross != null) s.AddLast(stOOLCross);

        var stOOLLastLayer = SolverOOLLayer.Solve(s.LastCube);
        if (stOOLLastLayer != null) s.AddLast(stOOLLastLayer);

        var stPLLCorners = SolverPLL.Solve(s.LastCube);
        if (stPLLCorners != null)
        {
            s.AddLast(stPLLCorners);
        }

        return s;
    }
}
```

Metoda zna izračunati algoritem za vsako možno začetno stanje kocke, povprečni čas iskanja rešitve pa je približno 100ms.

Poglavje 5

Sklep

Pri diplomskem delu smo uspešno rešili zastavljene cilje, predvsem:

- Struktura kocke: eden izmed ciljev je bil predstavitev Rubikove kocke v kontekstu objektno usmerjenega programiranja. To smo rešili s kreiranjem ustreznih razredov in relacijami med njimi.
- Izdelava programskega vmesnika (API): vsi potrebni razredi in metode za manipulacijo s kocko so v knjižnici RubiksCore, kjer je programski vmesnik jasno definiran, zato se lahko knjižnica uporabi pri drugih projektih v .NET okolju.
- Predstavitev kocke v 3D okolju: končna aplikacija zna kocko prikazati v 3D okolju in omogoča manipuliranje kocke s pomočjo miške. Animacije rotacij uporabniško izkušnjo še izboljšajo.
- Algoritem za rešitev kocke: aplikacija omogoča vnos poljubne kocke preko 3D vmesnika, izračun algoritma za rešitev in prikaz celotne rešitve v posameznih korakih.

Trenutno je glavna pomanjkljivost aplikacije ta, da za normalno delovanje potrebuje podatkovni strežnik SQL Server. To je mogoče precej enostavno rešiti tako, da vse potrebne algoritme iz podatkovne baze prenesemo v drugačen format, na primer v datoteko na trdi disk. Vsebino bi pri zagonu aplikacije

shranili v pomnilnik v ustrezno strukturo in jo nato uporabili pri izračunu rešitve.
Končna aplikacija sicer izpolnjuje vse podane zahteve, se je pa pri razvoju pojavilo več idej za nadgradnjo:

- Doda se lahko več metod za izračun rešitve kocke. Ker imamo vso potrebno kodo že na voljo (razrede za predstavitev kocke in rešitve ter metode za rotacije), se lahko pri tem osredotočimo zgolj na metodo, ki jo želimo implementirati.
- Z manjšimi spremembami grafičnega okolja bi lahko aplikacijo priredili za mobilne naprave, ki podpirajo .NET ogrodje. Potrebno bi bilo prepisati dogodke, ki jih zdaj aplikaciji posreduje miška, v dogodke, ki jih podpirajo mobilne naprave.
- V aplikacijo bi se lahko dodala baza algoritmov z ustreznimi animacijami.

Literatura

[1] Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner, *C# 2008*, Indianapolis: Wiley Publishing, Inc., 2008

[2] Adam Nathan, *Windows Presentation Foundation*, U.S.: Sams Publishing, 2007

[3] Ray Rankins, Paul Bertucci, Chris Gallelli, Alex T. Silverstein, *SQL Server 2005*, U.S.: Sams Publishing, 2007

[4] (2012) Speedsolving.com Wiki. Dostopno na: http://speedsolving.com/wiki/

[5] 2013 Rubik's Cube Solution. Dostopno na: http://cubefreak.weebly.com/